METHOD FOR HIDING LATENCY IN A TASK-BASED LIBRARY FRAMEWORK FOR A MULTIPROCESSOR ENVIRONMENT

TECHNICAL FIELD

The invention relates generally to multiprocessor environments and, more particularly, to a task-based library framework for dynamic load balancing in a multiprocessor environment, and to a method of latency hiding in this framework.

BACKGROUND

A multiprocessor system executes a program faster than a single processor of the same speed because the multiple processors work simultaneously on the program. In such a system, programs are subdivided into tasks and the resultant tasks are assigned to processors. To take maximum advantage of a multiprocessor system, it is necessary to have all processors working simultaneously when any is. Load balancing is the attempt to evenly divide the tasks or workload among the processors. In traditional methods of load balancing, each processor has a queue of tasks. A central task-distributor assigns each new task on arrival to the queue for a processor. Some standard methods are round-robin, random, and assessment of how busy the processors are. In standard methods, the central distributor tries to predict the future to assess how long each processor requires to complete the tasks in its queue. The distributor's assessment is not always accurate, however. As a result, some processors sometimes have long queues of tasks while others are idle. Consequently, execution of the program is delayed.

In addition, the central distributor may be heavily burdened with the distributing of the tasks to the

processors. Finally, there may be a delay in latency in task-loading or taking a task from the central distributor and loading it into a processor.

Therefore, there is a need for a method of load balancing in a multiprocessor system, that more evenly balances the load among the processors than traditional methods, does not burden the central distributor, and reduces the latency in task-loading.

SUMMARY OF THE INVENTION

The present invention provides a task-based library framework for load balancing using a system task queue in a tightly-coupled multiprocessor system. The system memory holds a queue of system tasks. The library processors fetch tasks from the queue for execution.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIGURE 1 schematically depicts a tightly-coupled multiprocessor system with a task-based library framework and library processors;

FIGURE 2 illustrates a library processor with double buffer for holding tasks;

FIGURE 3 depicts a flow diagram of the subdivision of tasks into subtasks and the assignment of the subtasks to the library processors; and

FIGURE 4 depicts a flow diagram which illustrates the loading of tasks onto a library processor.

DETAILED DESCRIPTION

In the following discussion, numerous specific details are set forth to provide a thorough understanding of the present invention. However, it will be apparent to those skilled in the art that the present invention may be practiced without such specific details. In other instances, well-known elements have been illustrated in schematic or block diagram form in order not to obscure the present invention in unnecessary detail.

It is further noted that, unless indicated otherwise, all functions described herein may be performed in either hardware or software, or some combination thereof. In a preferred embodiment, however, the functions are performed by a processor such as a computer or an electronic data processor in accordance with code such as computer program code, software, and/or integrated circuits that are coded to perform such functions, unless indicated otherwise.

Referring to FIGURE 1 of the drawings, the reference numeral 100 generally designates a tightly-coupled multiprocessor system with a task-based library framework. The system 100 comprises a system kernel 102, a system memory 104, and a number of library processors, 108, 110, 112, 114, and 116. The ellipsis indicates that the system 100 can comprise additional library processors. The system memory comprises a queue of tasks 106 to be assigned to the library processors (library task queue 106). Each library processor has access to the library task queue 106. When tasks arrive at the system kernel 102 for processing, they are subdivided into subtasks and placed into the library task queue 106. The library processors 108, 110, 112, 114, and 116 fetch the subtasks from the library task queue 106.

Referring to FIGURE 2 of the drawings, illustrated is a library processor 200. It comprises a kernel 202, and a

local memory 204. The local memory comprises buffers 206 and 208. When the library processor fetches a task from the library task queue 106, it loads it into one of the buffers 206 or 208. The library processor 200 can execute a task contained in one of the buffers while it is loading a task into the other buffer. As a result, the latency of task-loading is avoided. In addition, the library processor 200 shares in the work of the distribution of tasks from the library task queue 106. Thus, a heavy burden on a centralized task distributor is avoided.

Referring to FIGURE 3 of the drawings, illustrated is a flow chart of the subdivision of tasks into subtasks and their distribution to the library processors. An incoming task arrives at the system kernel 102. A thread of the main process (or a different process) submits the task to the system kernel 102 and blocks on a semaphore until the task is finished, when all the subtasks are finished and the semaphore is unblocked by the system kernel 102. In a server environment, the number of the processes is large enough to keep all the library processors 108, 110, 112, 114, and 116 busy and thus achieve the optimal through-put.

The task is subdivided into subtasks, which are placed in the library task queue 106. The library processors 108, 110, 112, 114, and 116 fetch the subtasks from the library task queue 106 into their buffers 206 and 208, process the tasks, return the results to the library task queue 106, and mark the results done. The system kernel 102 will "poll" for the results and status of the set of related tasks. The data structure tracking subtasks is shared by the system kernel 102 and the library processors 108, 110, 112, 114, or 116 that work on the subtasks. To ensure that all the library processors 108, 110, 112, 114, and 116 are working, the

number of independent subtasks is larger than the number of available library processors.

For this method of subdividing tasks and distributing them to library processors to be effective, the multiprocessing system 100 must be tightly coupled. The time required for moving a task from the library task queue 106 to a library processor 108, 110, 112, 114, or 116 must not be substantially longer than the time to complete a task. Otherwise, there would be a delay while a task was being loaded in a library processor. One embodiment uses specially-designed communications channels to speed up the loading of tasks from the library task queue 106 to the library processors 108, 110, 112, 114, and 116.

Now referring to FIGURE 4, shown is a flow diagram which illustrates the loading of tasks onto a library processor. In step 402, the library processor kernel 202 checks the number of tasks residing in the buffer. If two tasks are residing, in step 408, the library processor kernel 202 prepares the execution environment for the first ready-to-run task. In step 410, the library processor kernel 202 passes control to the first ready-to-run task for execution. Upon completion of the task, the process returns to step 402.

If one task is residing in a buffer, in step 406, the library processor kernel 202 preloads a second task. Then, the process then goes to step 408. The new task from the library task queue 106 is loading while the old task is executing. As a result, the latency of loading is reduced or completely eliminated. Several mechanisms enable the simultaneous loading of a new task while the old task is executing. One such mechanism is a DMA mechanism that loads the new task. If there is no task in the library task queue 106 at step 406, the library processor kernel 202 executes

the task in the buffer by proceeding to steps 408 and 410.

If no tasks are residing in the buffer, in step 404, the library processor kernel 202 fetches a task from the library task queue 106 and returns to step 402. If there is no task in the library task queue 106, the process waits until there is a task.

Steps 404 and 406 are where the load balancing occurs. The library processors 108, 110, 112, 114, or 116 fetch tasks from the library task queue 106 in these steps. Since a library processor 108, 110, 112, 114, or 116 fetches tasks only when there is at most one task in the buffers, the load on a library processor is never more than two tasks, one of which is executing. As a result, the load is evenly balanced. No library processor 108, 110, 112, 114, or 116 ever has more than one task in its buffers 206 and 208 awaiting execution while another library processor 108, 110, 112, 114, or 116 is idle.

To assure synchronicity, some bookkeeping steps are needed, which were glossed over above. When a task is fetched from the library task queue 106 at step 404 or step 406, the library task queue 106 is locked, the task to be fetched is marked 'working', and the library task queue 106 is unlocked. When a task has been processed, at the completion of step 410, the library task queue 106 is locked, the result of the task is updated and the task marked done, and the library task queue 106 is unlocked.

In one embodiment, the library processors 108, 110, 112, 114, and 116 use DMA mechanisms to load the task. The DMA mechanisms all share the same synchronization scheme/atomic access to the library task queue 106, thus enabling the transfer of a task from the library task queue 106 to one and only one library processor 108, 110, 112, 114, or 116.

It is understood that the present invention can take many forms and embodiments. Accordingly, several variations may be made in the foregoing without departing from the spirit or the scope of the invention. The capabilities outlined herein allow for the possibility of a variety of programming models. This disclosure should not be read as preferring any particular programming model, but is instead directed to the underlying mechanisms on which these programming models can be built.

Having thus described the present invention by reference to certain of its preferred embodiments, it is noted that the embodiments disclosed are illustrative rather than limiting in nature and that a wide range of variations, modifications, changes, and substitutions are contemplated in the foregoing disclosure and, in some instances, some features of the present invention may be employed without a corresponding use of the other features. Many such variations and modifications may be considered desirable by those skilled in the art based upon a review of the foregoing description of preferred embodiments. Accordingly, it is appropriate that the appended claims be construed broadly and in a manner consistent with the scope of the invention.